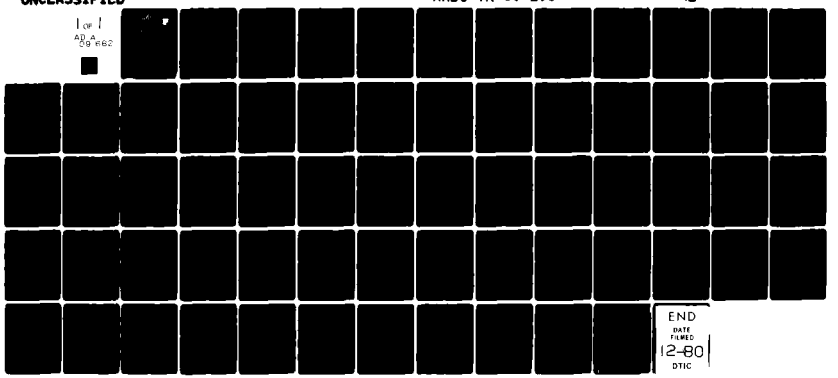
AD-A091 682 TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CA F/G 9/2
FORMAL SEMANOL SPECIFICATION OF ADA.(U)
SEP 80 P T BERNING F30602-79-C-0204
UNCLASSIFIED RADC-TR-80-293 NL



END
DATE FILMED
12-80
DTIC

LEVEL II

12

RADC-TR-80-293
Final Technical Report
September 1980

FORMAL SEMANOL SPECIFICATION OF ADA

TRW Defense and Space Systems Group

Paul T. Berning

Laboratory Directors' Fund No. LD9205C1

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-293 has been reviewed and is approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

SUBJECT TO EXPORT CONTROL LAWS

This document contains information for manufacturing or using munitions of war. Export of the information contained herein, or release to foreign nationals within the United States, without first obtaining an export license, is a violation of the International Traffic in Arms Regulations. Such violation is subject to a penalty of up to 2 years imprisonment and a fine of $100,000 under 22 U.S.C 2778.

Include this notice with any reproduced portion of this document.

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-80-293	2. GOVT ACCESSION NO.: AD-A091682
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): FORMAL SEMANOL SPECIFICATION OF ADA	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. Jul 79 – Jun 80
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): Paul T. Berning	8. CONTRACT OR GRANT NUMBER(s): F30602-79-C-0204
9. PERFORMING ORGANIZATION NAME AND ADDRESS: TRW Defense and Space Systems Group, One Space Park, Redondo Beach CA 90278	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101F, LD9205C1
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: September 1980
	13. NUMBER OF PAGES: 70
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Douglas A. White (ISIS)
This effort was funded totally by the Laboratory Directors' Fund

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Specification
Computer Programming Language
Ada
SEMANOL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarizes the performance and results of a contractual effort to develop a formal operational specification of the DoD common programming language Ada. The formalism used was that of the Semantics Oriented Language, SEMANOL. The design produced essentially covers the entire Ada language, ignoring only the low-level semantics of implementation dependencies. The SEMANOL system and its use in the specification of Ada language features of a type not previously addressed in SEMANOL

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

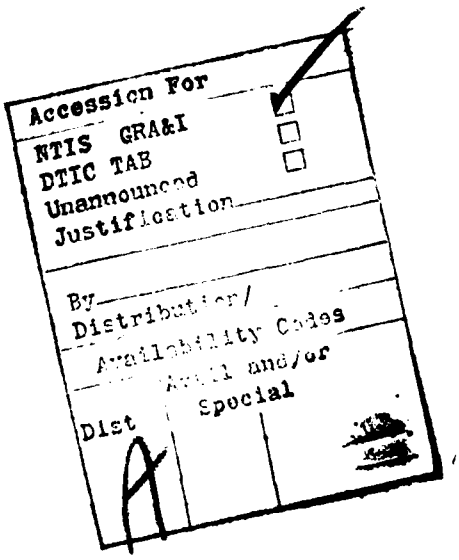
specifications of other languages is provided. The report is concluded with a list of problems discovered, in the design of the Ada language, as a result of the formal specification.

Abstract

This project was undertaken in order that a formal operational specification of the Ada programming language would be designed. The formalism used was that of SEMANOL, TRW's well established metalanguage for completely defining the syntax and semantics of programming languages. The design produced in this project essentially covers the entire Ada language, ignoring only the low-level semantics of implementation dependencies, since deferral of these was required by the conditions of this project. The SEMANOL design specifically provides an operational framework in which concurrent task execution can be defined. To support this form of parallel execution semantic definition, a very few extensions to the SEMANOL metalanguage were made; however, these extensions were done so as to leave SEMANOL fully upward compatible from the prior version of the metalanguage. The design provides both concrete and abstract syntax specifications, and includes the algorithm by which the concrete grammar is mapped to the abstract form. The semantics are given in operational terms, as is SEMANOL's purpose, and deal with overloading, generics, exceptions, all Ada types, etcetera, as well as concurrency. A design basis has thus been established from which a full, operational, SEMANOL specification of Ada can be constructed.

This report summarizes the performance and results of contract performance, while the accompanying report, "The Design of a SEMANOL Specification of Ada," presents the detailed technical products of this effort.

Contents



EVALUATION

This effort directly supports the Higher Order Language sub-thrust of the Software Cost Reduction thrust by providing two valuable details necessary for successful language development and control. These details insure a language in which errors, inconsistencies, and ambiguities have been removed, and a formal specification to provide implementors an exact definition of the language.

The goals of this effort, through the process of formal specification using SEMANOL, were to uncover deficiencies in the Ada language, and to design a formal SEMANOL specification for Ada. The information and design resulting from this process are to be available to the Air Force and Department of Defense in their programs to develop and control the Ada language.

Douglas A. White

DOUGLAS A. WHITE
Project Engineer

INTRODUCTION

This project was undertaken with the purpose of designing a SEMANOL specification for the Ada programming language. Since SEMANOL is a metalanguage meant to express formal specifications for programming languages, a SEMANOL specification of Ada can provide a precision to Ada definition that is missing in conventional prose descriptions. SEMANOL's operational nature also means that semantic details, even those related to language implementations, can be clearly identified, and implementation guidelines provided to those who must provide Ada processors and host environments. As Ada is a new language, even now undergoing refinement, a formal operational definition can also contribute to clarity of language design and understanding of that design. In particular, the use of SEMANOL can contribute to resolving issues of (1) unintended vagueness versus intentional omissions for implementation dependencies, (2) definition of conformance of programs and language processors to a standard, and (3) the classification of errors (e.g., run time or compile time). A SEMANOL specification of Ada can thereby contribute to Ada's acceptance on the wide scale that is now hoped for.

The SEMANOL system used in this project was developed by TRW over the past ten years. It consists of a theory of semantics based on traces, a metalanguage in which operational specifications can be written, and an Interpreter to process metaprograms written in the SEMANOL metalanguage. SEMANOL has previously been used to give definitions of JOVIAL(J3), JOVIAL(J73), CMS-2, SMITE and BASIC that were complete and Interpreter-tested to a substantial degree. Designs for SEMANOL specifications, complete with illustrative fragments of SEMANOL metaprogramming, have also been created for COBOL and Pascal. These past projects amply demonstrate SEMANOL's ability to fully describe a wide range of typical, frequently used, programming languages. Since Ada has much in common with these older languages, the semantic description techniques developed in past SEMANOL applications have often been adaptable to Ada. At the same time, Ada contained features that the programming languages treated before had lacked; the most notable new feature being the possibility of concurrent execution of parallel tasks.

Thus a major effort was made in this project to extend the SEMANOL theory and the metalanguage so as to provide a satisfying way in which to describe the semantics of concurrency. This was accomplished by creating a system of SEMANOL processors whose scheduling and interaction are determined by an element of the SEMANOL metaprogram that is analogous to a host operating system nucleus. The implementation characteristics of Ada concurrency are thereby made very visible. This was achieved without the need to make many changes to the SEMANOL metalanguage itself, and making these extensions so that the SEMANOL specifications written earlier remain valid (i.e., upward compatibility was maintained).

The results of this research activity were the successful production of a design for a SEMANOL specification of Ada and the development of the SEMANOL metalanguage extensions needed to support this form of specification. This design and the factors that led to it are given in a companion report to this one, "The Design of a SEMANOL Specification of Ada". The Design document contains the detailed technical output produced in performance of this project. The remainder of this Final Technical Report summarizes these results and provides an introduction to the SEMANOL system used for the Ada design formalization. A description of the methods used and the factors affecting project performance is also included, as are a few conclusions drawn from the conduct of this effort. The Final Technical Report is meant to give the sense of what has been done in this project without being unduly technically demanding; the interested specialist is urged to read the Design report for a full treatment of the issues involved in creating a formal SEMANOL specification of Ada.

PROJECT PERFORMANCE SUMMARY

This project was undertaken with the purpose of producing a design for a SEMANOL specification of Ada. The SEMANOL specification of Ada is then expected to be useful because it can compliment the conventional language descriptions by being more precise and complete than they and, by giving an operational meaning to Ada semantics, can clarify the nature of the implementation dependencies of Ada. The type of SEMANOL specification we believe would do this is described in detail in the accompanying Design Report. In that report, the design for a SEMANOL specification of Ada is presented along with a technical justification for the approach therein followed. In this section of the Final Technical Report, we will simply attempt to describe a few of the major points bearing upon our performance of this contract.

It should be realized that designing and writing formal specifications is always a difficult task. Since it involves making precise what is imprecisely explained in a conventional programming language manual, a great deal of analysis must be done to decide just what the language being defined (e.g., Ada) is thought to most nearly be. The areas in which this hypothetical model are doubtful, where one is filling in gaps in the conventional manual or resolving apparent conflicts therein, can be difficult to recognize and extremely difficult to resolve in a satisfactory way. The language model must then be expressed in the SEMANOL system. The major problem in this, a familiar one to programmers but exacerbated here, is to create a SEMANOL metaprogram that is clear, through its organization and style, to its human readers. To make complex ideas understandable is always hard, and Ada certainly presents this challenge. And while clarity is of paramount importance, SEMANOL metaprograms are expected to be executed by the SEMANOL Interpreter; thus the efficiency of the metaprogram is considered and alternate approaches may need to be evaluated. Writing the SEMANOL specification design of Ada has thus demanded technical competence and individual perseverance. We believe the results will be useful.

The approach followed in conducting this project was essentially one of (1) analyzing Ada, (2) deciding what parts of Ada could be adequately treated by traditional SEMANOL techniques, (3) and developing new SEMANOL methods for the novel features of Ada. A guiding principle in our conduct of this project was the desire to minimize changes to the SEMANOL metalanguage; a corollary to that was the wish to maximize the use of methods that were successful in past applications of SEMANOL. Since metalanguage changes were so few and localized, the metalanguage remains upward compatible with the existing version; older SEMANOL metaprograms will still run without change. Since proven methods were so often adaptable to Ada, our analysis could be focused upon the newer features of Ada and overall confidence in the resulting design increased.

The technical performers for this project were Mr. Frank Belz, Dr. Edward Blum, and Mr. Dennis Heimbigner. All have been involved in many past SEMANOL projects and have been responsible for SEMANOL's development and extension; of course, Dr. Blum is the inventor of SEMANOL. All had also followed the Ada development process and participated in the various language and requirements reviews that were part of this Ada development. An exceptionally fine group of computer scientists, extraordinarily well versed in the appropriate technologies, have thus been responsible for TRW's construction of the design for the formal Ada specification.

The Ada language for which the design of a SEMANOL specification was developed reflects our interpretations of the "Preliminary Ada Reference Manual" and the "Rationale for the Design of the Ada Programming Language." Our understanding of Ada was also influenced by reading the many Language Issue Reports submitted, by Mr. Belz's participation as a Distinguished Reviewer and, to a much lesser extent, by an analysis of the many reports that have been written about Ada. The formal denotational specification was also a useful source document for this project. It provided an alternate type of definition of Ada, and so complimented the conventionally presented definitions of the Reference Manual and Rationale. Since all three documents were written by the Ada design team, they could be assumed to be three views of essentially the same language; it was thus possible for us to use all three when searching for Ada. Our design

report also tends to follow the organization of the denotational specification in that it presents the SEMANOL design in an Ada feature-by-feature basis generally, instead of by SEMANOL metaprogram sections. The two approaches to formal semantics are thereby more readily comparable.

While the denotational specification was helpful, it should be remembered that it ignores Ada concurrency altogether (because of its difficulty). The SEMANOL design provides a solution to this complex technical problem. The SEMANOL design is not just a restatement of the denotational specification, but one that goes beyond it to formalize a greater part of Ada.

It should be remembered that Ada is a new programming language and one that is still undergoing design changes. Thus the analysis done in this contract has had to cope with the fact the language itself is changing. Also, since the language is a new one, the defining documentation for it is somewhat incomplete and occasionally ambiguous. Thus the major source documents from which this work was done were themselves somewhat imperfect. On the other hand, Ada is also a language of great interest to many people, a large number of whom are involved at present in analyzing and evaluating the language. Many of these have discovered problems in the Ada reference manual (as have we), and by so doing have aided us in our analysis of that same manual. The Language Issue Reports have often reported definition problems that we had also discovered, and by so doing have relieved us of the necessity of preparing duplicate reports. So while the newness of the Ada language has complicated our task, the fact that the language is being carefully scrutinized by so many others has at the same time simplified our task.

Our analysis has also been assisted by the fact that Mr. Frank Belz has served as a Distinguished Reviewer for the Ada programming language. In this capacity, Frank has attended many meetings in which the various language design and specification issues were discussed. His participation has thus assured us of being conversant with the latest prospective changes to Ada. In this way, we have been able oftentimes to anticipate changes in Ada and to include these prospective changes in the work we have performed. Mr. Belz's participation

has also meant that the results of TRW's analysis of Ada have been directly presented to the committee. So the results achieved in performance of this contract have already been injected into the Ada development process. Frank's participation as a Distinguished Reviewer was not covered by this contract itself, but it has, nevertheless, contributed to making our performance better than it would otherwise have been and in making some of the results of this project immediately useful.

The major product of this contract is the design document for the SEMANOL specification of Ada. In it will be found our approach to defining the semantics of Ada in a formal operational manner. The specification that has been created is complete except for certain omissions to be described later, but does vary in the level of detail in which the various language features are described. The document does contain a complete concrete grammar for the Ada programming language and a complete abstract grammar. The document also contains a specific algorithm by which the concrete grammar can be matched to the abstract grammar. The grammars and the explicit mapping algorithm between them do constitute a contribution to the Ada development. To our knowledge, this has not existed prior to our performance upon this contract.

While the grammar of Ada has been treated very completely and in thorough detail, the semantics of Ada have been considered somewhat less fully and less evenly. As suggested before, those parts of Ada which have previously been dealt with in SEMANOL, such as evaluation, have not in all cases had their semantics worked out in detail in performance of this contract. We essentially rely for the design on citing the analogy of past performance and pointing out how these previously developed methods can be applied to Ada. It is largely in the area of features to which SEMANOL has not been applied that our emphasis has been given and that the design report itself largely directs its attention. Thus an extensive discussion of the semantics of concurrency will be found there.

Certain features described in the Ada Reference Manual have been disregarded in the SEMANOL design as being beyond the scope of Ada semantics. They are

important to Ada users, but their appearance in a formal Ada definition would be unconventional and unhelpful. These exclusions are as follows:

1. Machine code insertions. Ada permits the in-line insertion of machine code. While the specification of the semantics of machine code for some hypothetical machine could readily be accomplished in SEMANOL, there was certainly no reason to do so in this research-oriented project.

2. Foreign languages. Ada is also meant to provide facilities by which subprograms written in a language other than Ada can be connected with Ada programs; that is, calls to subprograms in another language are to be supported by Ada. As with machine code insertions, this too could be done with SEMANOL. One could, for example, choose to consider BASIC, for which a SEMANOL specification already exists, and thereby support this option. However, as in past SEMANOL projects, we have deliberately chosen to consider the formal specification of subprograms written in a different language as beyond the domain of Ada.

3. Library facilities. We have always considered the possibility of library support facilities to be a compiler-related feature of a host environment, and not something to be thought of as an intrinsic part of the semantics of a programming language. That is, the definition of a library is beyond the scope of a formal programming language specification. This view applies to Ada as well.

   Now SEMANOL could provide a library definition. However, many of the characteristics of the total library facility, and most of the details, would only be defined by the host environment in which a given Ada compiler operates. Thus a formal definition of a library could only be suitable for an arbitrary conjectural situation. It is also a good deal of effort, even with SEMANOL, to implement a library creation and maintenance facility. For

these reasons, this is an Ada feature whose specification design is best ignored, or at least deferred.

4. Pragmas. Pragmas are meant to supply information to a given Ada compiler, and some pragmas are considered language defined. However, even the language defined pragmas (e.g., SUPPRESS) are expected to be without semantic effect, and so can be disregarded in a SEMANOL specification.

A few other aspects of Ada were largely ignored in the design process as being low level implementation dependent semantics best considered later. These include the following:

1. Approximate arithmetic. If a SEMANOL specification is to be made complete to the degree that it is to be operational, some sort of arithmetic must be provided for. Thus the way in which real numbers are to be represented and the way in which arithmetic is to be performed upon these approximations must needs be defined. We have defined this kind of arithmetic in past projects, and have generally attempted to do so in a parametric way in order that the specification given could be as general as possible. Nevertheless, this is a very low level part of any programming language specification; since our design effort here is generally being conducted at a higher level, it is simply thought unsuitable to attempt to provide details for approximate arithmetic at this time.

2. Input-output. Ada does not contain explicit input-output operations, nor does it presume any specific interface through which Ada programs can communicate with external devices; thus input-output semantics per se are outside the scope of Ada. The details of input-output, which is assumed to be possible, are to be entirely implementation determined. This means any effort to include them in a formal definition must invent an external "world," possibly in the form of an event-driven operational model. While SEMANOL can do this, it has no special facilities

for such modeling. For these reasons, we have made no effort to formulate a generalized environmental framework in which Ada programs would execute.

Some input-output packages that are to support Ada programs are given in the Ada Reference Manual. LOW-LEVEL-IO is essentially implementation defined, and its modeling would present the problems already noted. INPUT-OUTPUT and TEXT-IO are somewhat more abstract, and are similar in function to input-output features modeled previously with SEMANOL. TEXT-IO features can be explained with the #INPUT and #OUTPUT operators of SEMANOL, while INPUT-OUTPUT can make use of SEMANOL sequences. The higher-level forms of Ada input-output do lend themselves to a relatively straightforward definition in SEMANOL. The uncertain relationship of input-output to Ada, the anticipated major changes to Ada in this area, and the fact that dealing only with higher-level input-output is still a substantial task, caused us to generally disregard input-output altogether in our design. This could be reconsidered in a later implementation phase (where some input-output facility would probably become necessary).

3. Interrupts. Interrupts are related to low level input-output, and their nature and timing are likewise implementation determined. Our design does model the receipt and handling of interrupts, but not the simulation of their generation. The generation of interrupts, like input-output modeling, could be provided by SEMANOL's ability to use external functions. However, such functions are so closely related to the host environment and Ada program application that the design of any interrupt functions was thought best delayed until an implementation phase. (We should also note that SEMANOL metaprograms are not expected to supply test-bed environments.)

4. Detailed storage model. Ada's address-specification is used to position code at a particular machine address or to define an address as the location of a variable or hardware interrupt

entry transfer. Its semantics are inherently implementation defined, having a significance dependent upon the particular machine organization used and that machine's interface to external devices. Such machine sensitive low level details are being ignored in this design effort. Presumably, the address-specification would be used to support low level input-output as realized with LOW-LEVEL-IO and similar packages.

In a similar way, length-specification, packing-specification, enumeration-type-representation and record-type-representation allow one to control the internal structure of data elements. They can easily be modeled with the use of SEMANOL, but they lack semantic significance and so have been disregarded in this project.

While the design has passed over some features of Ada, at least for now, it has also included aspects of Ada that are generally ignored when using other formal methods. Thus the issues of separate compilation and real-time clocks have been addressed. Separate compilation is a feature of a programming language that allows a program to be broken into parts, compilation units in Ada, such that parts translated separately by a compiler can be combined somehow into a single program for execution (possibly by a linking loader). The units needed to make a compilation unit whole for execution are maintained in a library; so the issues of separate compilation and libraries are themselves closely related. Our design for a formal specification of Ada omits specification of a library for the reasons given before, but does include enforcement of the visibility rules of Ada that support separate compilation. Thus the SEMANOL specification design expects the entire program to be given as a body by #GIVEN-PROGRAM, and then checks the text for correct unit ordering. Separate compilation implications are therewith included in our design.

While we had originally thought to disregard this particular feature of Ada, we have generally included facilities for modeling the behavior of clocks. Thus the time of day and delay intervals are provided for in the SEMANOL

specification of Ada. There is no pretense, however, that this "time" has any particular significance with regard to execution rates.

Besides including these unusual features, the SEMANOL design provides a formalism with which to explain the semantics of parallelism. We believe that this development of a semantic framework in which parallel execution can be explained is a noteworthy contribution of this project. Methods of formal semantic description have often been forced to disregard the semantics of concurrent execution. The approach we have developed is given in detail in the accompanying design report, where it is accompanied by the technical background needed to compare this approach with others that have been attempted. The approach is an operational one, as expected with SEMANOL, that depends upon the construction of an operating system kernel, with the kernel being described by the SEMANOL metalanguage. Essentially, a set of SEMANOL processes having the characteristics of physical processors are defined which, in turn, are used to interpret the concurrent Ada tasks. There is no one-to-one correspondence between SEMANOL processors and Ada tasks and, indeed, the number of SEMANOL processors is a specification parameter. The flexibility afforded by this approach means that it is well able to illustrate the practical problems of implementing the Ada language in the face of concurrent task execution. It is a pragmatic approach but, nevertheless, it does have a theoretical grounding as explained in the design document.

This approach to parallelism retained much of the control structure used in past SEMANOL applications for interpreting a single execution sequence. This traditional control structure is applied to each Ada task with SEMANOL being extended so that multiple concurrent SEMANOL processes can be applying this structure to multiple Ada tasks. In this way, the execution of concurrent Ada tasks can be simulated much as they might occur on a multiple processor host computer. The key decision in this design process was the decision to provide SEMANOL itself with concurrent metaprogram execution abilities. Had it been considered desirable to model Ada concurrency by use of a single SEMANOL processor, the SEMANOL language probably would not have changed. Although the extensions to SEMANOL were not strictly necessary, since Ada concurrency could be modeled by a use of a single processor SEMANOL metaprogram (just as Ada

programs containing concurrent tasks could presumably be executed on a single processor computer system), the use of the existent non-parallel SEMANOL system seemed unlikely to lead to a specification that would give its readers a clear picture of the implications of parallelism as it occurred in Ada. To capture Ada parallelism it simply seemed necessary to use a form of SEMANOL parallelism. We believe that the control model developed is one that reveals some of the implementation issues that arise in supporting Ada parallelism.

It is particularly welcome to note that the extension of SEMANOL so as to provide support for this type of concurrent task modeling involved very little change to the SEMANOL metalanguage. A primitive capable of initiating concurrent SEMANOL task processors (i.e., #CO-COMPUTE) and two primitives corresponding to integer semaphores (i.e., #P and #V) were all that were required. In addition, these changes were made in such a way that past SEMANOL specifications continue to be valid; that is, upward metalanguage compatibility has been maintained. Old SEMANOL specifications (e.g., BASIC and JOVIAL) remain valid.

Although the operational nature of the SEMANOL concurrent-process model introduces some degree of overspecification, being in effect an implementation of Ada, the act of constructing it has pointed up certain difficulties and subtleties that exist in Ada and which will cause difficulty in any implementation. For example, the unblocking of a task blocked on a select which involves both an accept and a delay was found to require careful attention to avoid deadlock, with rather subtle analysis required of mutual exclusion and synchronization techniques and choice of correct unblocking alternatives.

We should note that it had been our original intention to use the Ada Translator program as a guide to resolving questions arising from ambiguity in the Reference Manual. The idea was that Ada test programs illustrative of the issue would be written and submitted to the Translator, and an analysis of the Translator results would then determine whether a certain usage was legal or not. In this way, the Translator could help resolve questions about the Reference Manual. Unfortunately, we found the Translator did not work well when we used it early

in the contract performance period. In general, it rejected our programs even though later consultation with the Ada language designers revealed the programs should have been accepted (e.g., the assignment of aggregates to multi-dimensioned arrays was considered illegal by the Translator when it should have been legal). Other Translator users reported similar problems. We thus concluded the Translator was an unreliable defining mechanism for Ada, and so could not provide the resolution of ambiguities that we had hoped it might; therefore, we abandoned efforts to use the Translator in this contract effort. Because of the Translator's failure in our demanding application, we derived no part of our design for the Ada specification from definitions implied by the Translator. (Since we weren't testing the Translator we, naturally, found no conflicts between the Translator and the Ada Reference Manual.)

THE SEMANOL SYSTEM

The term SEMANOL commonly is used to refer to the metalanguage of that name. But in a broader sense, it also includes a philosophical viewpoint of semantics, an approach to writing specifications in the metalanguage, and an Interpreter for the metalanguage that creates an operational system. All of these aspects are considered in what follows.

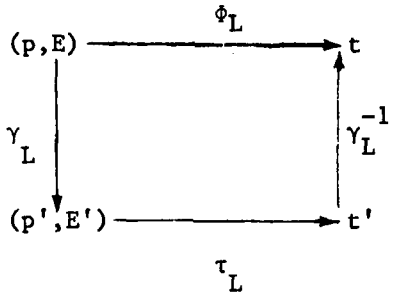
The general approach of this section is to discuss SEMANOL as it had evolved prior to the performance of this contract, and then to point out how it has been affected by performance of this contract. In the case of the metalanguage extensions, a separate subsection is given; for the theory, specification structure, and Interpreter presentations, the influence of Ada is given as an integral part of the text. It is intended that this section provide an introduction to SEMANOL, suggest how SEMANOL can be used to express a formal definition for Ada specifically, and show where the SEMANOL metalanguage and methodology have been affected by this application to Ada.

The SEMANOL Concept of Semantics

The SEMANOL concept of semantics of programming languages views a programming language, L, as a set of structured strings called programs. Each program, p, in L defines an algorithm, $\hat{p}$, for computing a function, $\hat{f}$. (Notation: When x is used to denote some expression, $\hat{x}$ is used to represent the thing denoted by x.) The function $\hat{f}$ and the algorithm $\hat{p}$ are both built up out of the operations and relations in a collection of data types associated with L. To describe $\hat{p}$ (or $\hat{f}$), a program p must contain constants and variables which denote the elements in the data types and which are combined with operation symbols into expressions and statements. The semantics of L tell us how to obtain $\hat{p}$ from p; that is, it tells us how, given p and some input data, E, which serve as initial values of some of the variables in p, to get a trace, t, of the computation produced by $\hat{p}$. (The value, $\hat{f}(\hat{E})$, of the function $\hat{f}$ is denoted by some constant which is part of the trace.) Thus the semantics of L is an operator, Φ_L, which maps such pairs

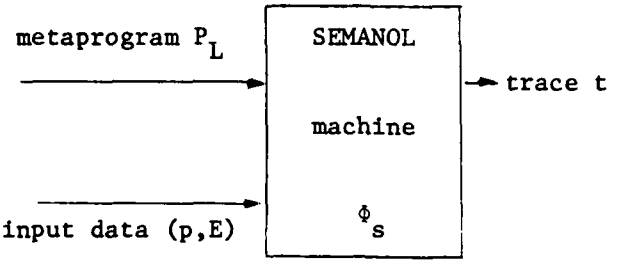
(p,E) onto traces. Φ_L is called the semantic operator of L.

A SEMANOL metaprogram, P_L, for language L describes the semantic operator Φ_L. It does this by specifying a meta-algorithm which produces a meta-trace from which the trace t can be extracted. We can depict how the meta-algorithm does this with a schematic diagram as follows:



Here we see Φ_L analyzed into three factors. The mapping γ_L converts the original program text p and input data constants E into representations, p' and E', in terms of SEMANOL data types. γ_L is described in a section of the metaprogram P_L involving lexical and syntactic analysis of the text p and the data E. This can also be thought of as a description of compile-time processing of p. The mapping τ_L produces a SEMANOL meta-trace, t'. τ_L is described in the semantic section of P_L in terms of SEMANOL operations and relations which are used to simulate the operations of L. The meta-trace t' is actually produced by executing the command section of P_L. Such execution follows the semantics, Φ_s, of the SEMANOL metaprogramming language, which is based on certain standard well-understood programming concepts. The meta-trace t' is essentially an expansion of t in which each operation of L is built up from SEMANOL operations. Finally, the trace t is recovered from t' by applying the inverse conversion mapping, γ_L^{-1}, which transforms SEMANOL constants back to the constants of language L.

The semantic operator of SEMANOL, Φ_s, can be thought of as a SEMANOL machine. The metaprogram P_L can be loaded into this machine together with a pair (p,E). The machine will execute the various sections of P_L and produce the trace t as output. Pictorially,



This view reflects the actual structure of the current SEMANOL implementation as shown in Figure 1. The SEMANOL Interpreter simultaneously defines the semantic operator Φ_s for SEMANOL and, in conjunction with the Multics system, incarnates the SEMANOL machine. In the same way, the SEMANOL description of programming language L simultaneously defines the semantic operator Φ_L for L and, in conjunction with the SEMANOL machine, incarnates an L-machine.

For the programming languages to which SEMANOL had been applied prior to this project, the trace was a single ordered sequence of operations; that is, parallel execution did not exist for these languages. Ada is different in that parallel task computations are an explicit part of the language, and the effects of this parallelism on program results (e.g., with regard to synchronization and orderly access to shared variables) are an explicit result of Ada program execution rather than implied by Ada semantics. In a sense, Ada is a relatively low level language when it comes to defining control flow; thus the details of parallelism must appear in the formal SEMANOL specification.

For Ada, the trace t (and its analogous t') will no longer be totally ordered when parallel computation occurs. In the face of parallelism, the semantics of Ada do not define the relative ordering between steps of parallel task activations; it is only at points of explicit synchronization that order is imposed. Thus each branch of a parallel computation is associated with an ordered sub-trace, but there is no accepted unique way in which these parallel sub-traces may be merged into the single trace assumed earlier for SEMANOL. The trace now becomes a collection of parallel trace chains, one for each concurrently active task, that are connected at points of task synchronization. Each step in a task trace chain is ordered by an earlier-than relation, as are

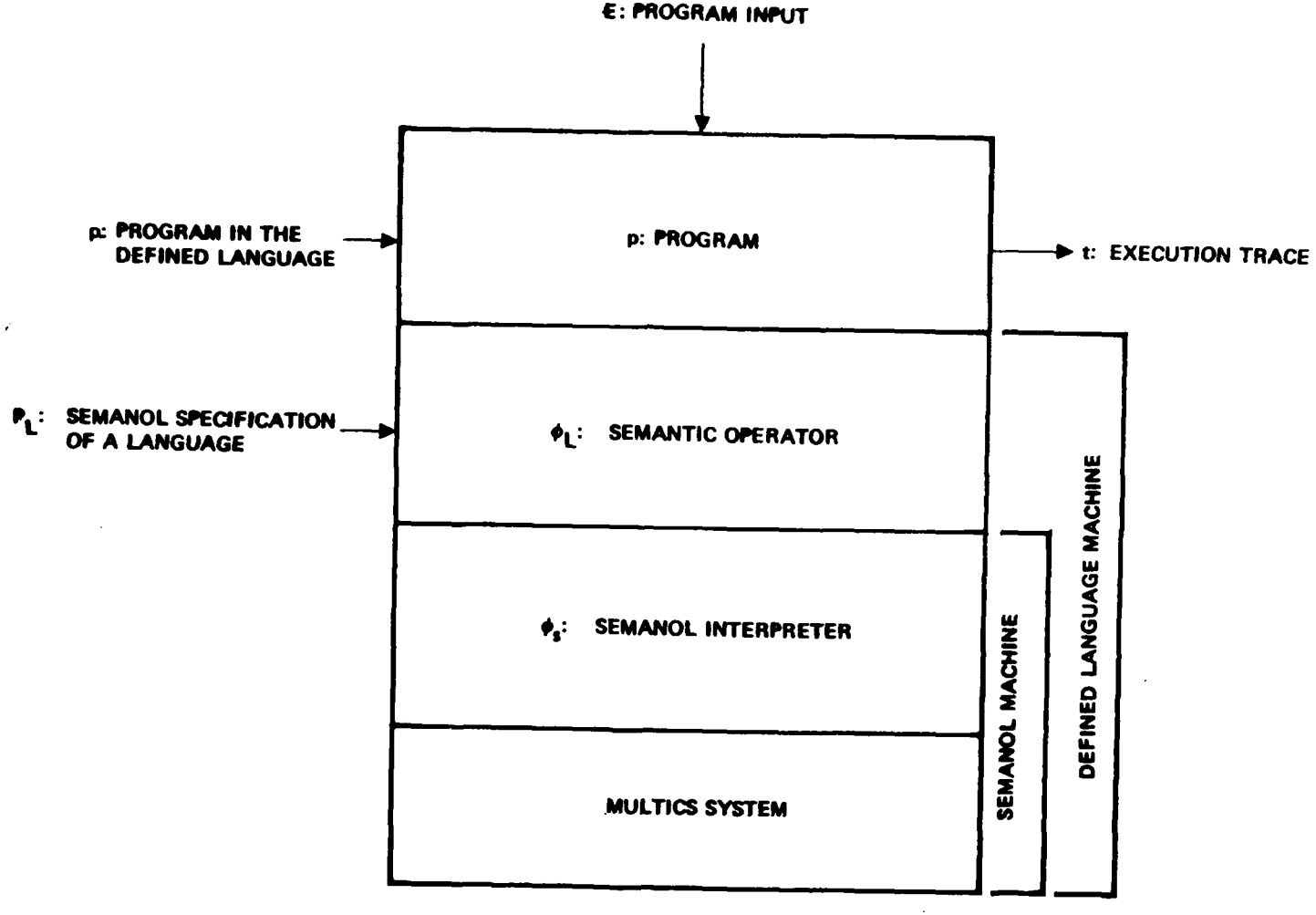


Figure 1: The SEMANOL Machine in Practice

the connecting synchronization steps (one thus has a semilattice). It is this new view of a trace that then guides the development of the design of a SEMANOL specification of Ada.

The Structure of a SEMANOL Specification

A formal SEMANOL specification of a programming language is a metaprogram, P_L, for processing a source language program text written in the programming language being defined. The algorithm expressed by the SEMANOL metaprogram describes a way in which the intended effect of executing any program in the defined language can be realized. That is, the algorithm is an interpretive definition of semantics or, alternatively viewed, the metaprogram describes an interpreter for the defined programming language. In what follows, we attempt to briefly describe this type of metaprogram through emphasizing the nature of its control flow logic, while only touching upon the data structures and representations involved. While the presentation is rather general, it does correspond to the design developed for Ada; indeed, those parts of the metaprogram whose treatment for Ada differs markedly from that used with other languages in prior contracts are noted. The changes in approach adopted for Ada are thus made clear.

A general approach to writing metaprograms has evolved through performance of past projects, and most of this general approach was retained when dealing with Ada. This approach is suggested in Figure 2. The left side of Figure 2 shows the transformations which are made to the representation of the input program text as interpretation is performed. The central block diagram is a simple flowchart showing the series of processing steps that the SEMANOL metaprogram typically causes to occur. This flow is a consequence of having an operational specification method. The right side of the diagram reflects the static structure of a SEMANOL metaprogram and the way in which the various statement groups are related to the specification logic. Observe that the SEMANOL metaprogram for Ada, as is customary with SEMANOL usage, expects programs in L to be given in terms of their concrete syntax; the metaprograms are thus thorough. This metaprogram structure is considered in more detail in what follows.

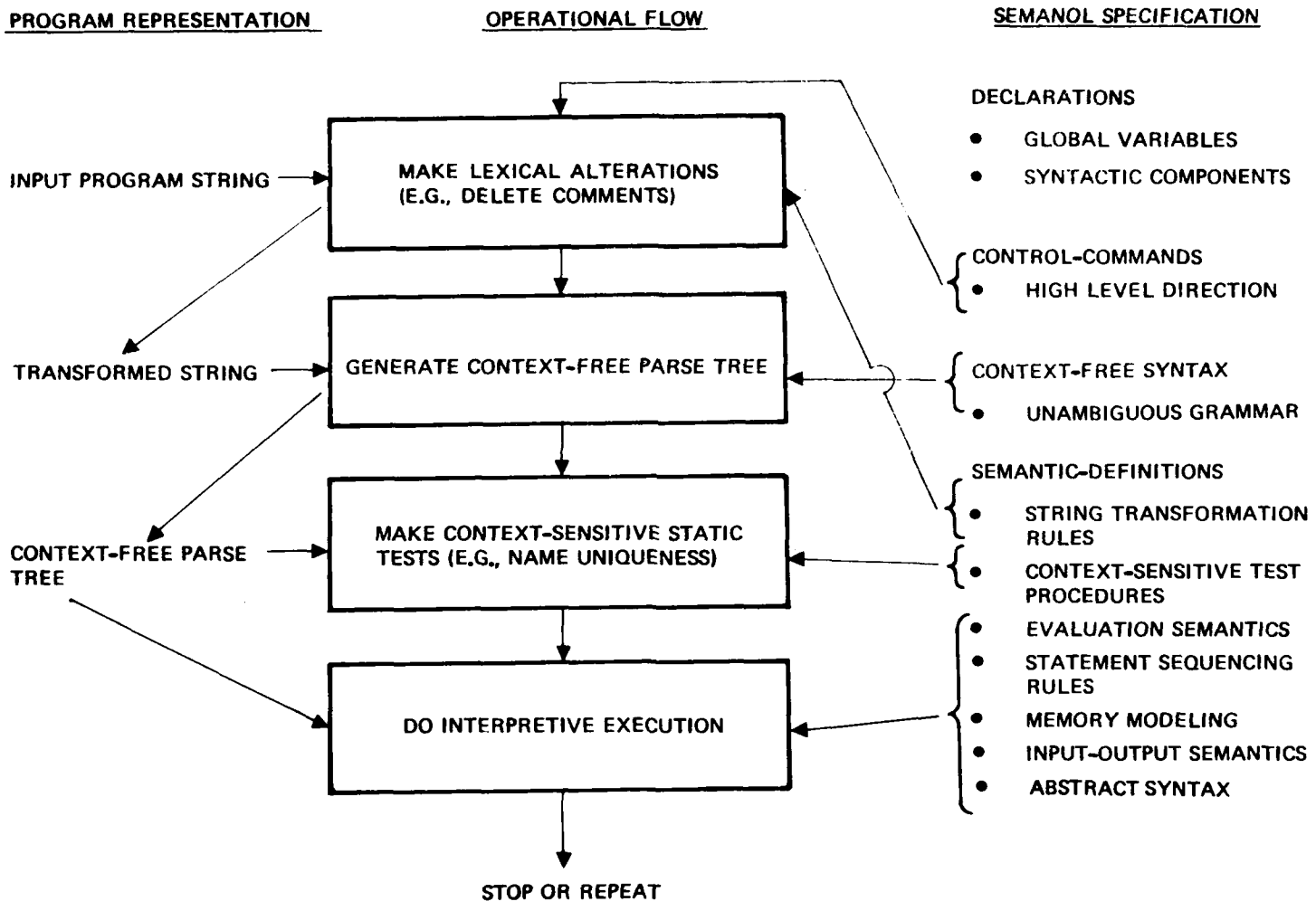


Figure 2: SEMANOL Specification Organization

The declaration section of a metaprogram is composed of SEMANOL statements that identify SEMANOL global variables and syntactic components. Very few global variables are normally used in a SEMANOL metaprogram; those of greatest importance are used to describe the control semantics of the defined language, and some are explained later. Syntactic components, on the other hand, are declared in generous number as an optimization measure. They declare the names of those SEMANOL definitions that produce constant values for a given parse tree argument, and so inform the Interpreter program to perform the computation of these functions only once for a given argument. The computed value is then associated with the argument node on the parse tree; later invocations of the function for that node need only retrieve the saved value rather than perform the computation. The definitions of the abstract grammar are the prime examples of syntactic components.

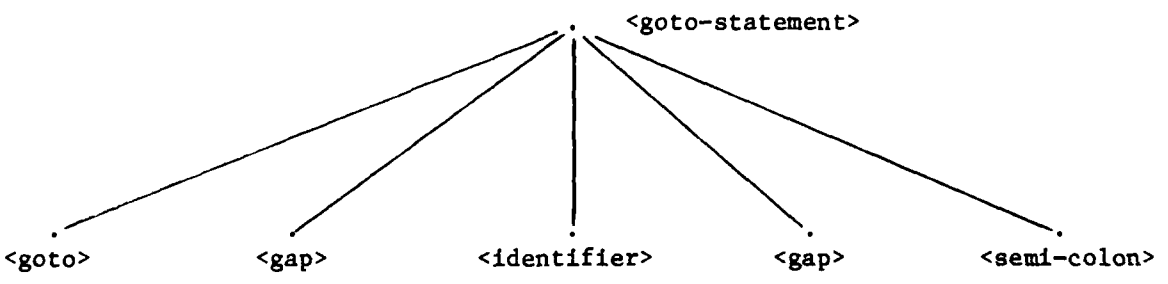
The control-commands section of a SEMANOL metaprogram is conventionally composed of a few SEMANOL commands. These commands are executed sequentially and correspond to those available in conventional programming languages. Interpretation of the SEMANOL metaprogram begins with the first statement of the control-commands section. These statements impose the overall control, as suggested in Figure 2; thus this section plays the role of a very high level main program.

The semantic definitions section of the SEMANOL metaprogram contains the detailed operational functions needed to define a programming language. The first operational step invoked in describing a programming language is to define the lexical transformations that the language includes, if any. These transformations may cause the source program to be altered as required for string substitutions or for line format considerations of the language being defined. They essentially are meant to put the program text into the form defined by the supporting context-free grammar for the subject language. In the case of Ada, this specification step will eliminate comments, change lower case letters to upper case, and map the full ASCII character set to the basic Ada one. This lexical transformation step is accomplished by parsing the text with respect to a simple context-free lexical grammar that identifies tokens,

gaps, and line terminators (including comments); collecting the items to be retained into a list; putting each list element into its cannonical string representation form; and reforming the list into the new text string. The transformed program text then becomes the basis for further specification.

Following the textual alterations, the transformed program text is parsed. This parse process is invoked by the SEMANOL operator #CONTEXT-FREE-PARSE-TREE and is directed by the grammar given in the context-free syntax section of the metaprogram. The product of this operation is a parse tree representation of the source program or one of two error conditions; the error conditions occurring if the grammar can lead to more than one parse of the given program (i.e., the grammar is ambiguous) or if the program cannot be parsed. The parsed representation reveals the structure of the program and so is a convenient basis upon which to formulate the later semantic description. The lexically transformed program text itself is retained as the terminal leaves of the parse tree. The SEMANOL syntactic definitions used to define the grammar look much like the productions of the usual treatments of such programming language grammars.

The context-free syntax used in this parse step to describe the programming language as written has come to be known as the concrete syntax, in contrast to an abstract syntax that provides a minimal program structure while still revealing the full semantic significance of a program. The abstract syntax strips away the stylistic flavor of a programming language in favor of exposing the semantic skeleton. SEMANOL employs both forms of syntax. For example, the concrete syntax of the Ada GOTO statement is given in SEMANOL by
#DF goto-statement => <goto> <gap> <identifier> <gap> <semi-colon> #.
The corresponding partial parse tree is



However, the only semantically important component of the goto-statement is the identifier part. An abstract syntax thus need only define a goto-statement as having an identifier element. In SEMANOL, the abstract syntax is implied by a set of selector and predicate functions. The goto-statement would thus lead to the creation of a selector function

```
#DF id-of(n) => #SEG 3 #OF n #IF n #IS <goto-statement> #.
```

and a predicate function

```
#DF is -goto(n) => #TRUE #IF n #IS <goto-statement>;
                => #FALSE #OTHERWISE #.
```

To assist in the derivation of the abstract syntax and to make explicit the connection between the abstract syntax and concrete syntax, we have annotated the SEMANOL concrete syntax for Ada in a standardized way illustrated by the following:

```
#DF goto-statement ": is-goto"
=> <goto> <gap> "id-of" <identifier> <gap> <semi-colon> #.
```

The two added quoted phrases are comments in the SEMANOL metalanguage, but they are formed so that they can readily be recognized and transformed into the selector and predicate definitions previously shown by a mechanical process. This process is now done manually, but we are weighing the desirability of extending SEMANOL so that this convention would become part of the metalanguage itself. In any event, by following this type of procedure, simplified in this example, we have created complete concrete and abstract grammars and, probably most importantly, have created a direct correspondence between the two. The understandability of the Ada specification is thereby improved.

Next comes the imposition of syntactic restrictions that cannot be expressed in a context-free grammar. That is, not all programs that can be parsed using the context-free grammar are legal programs in Ada. It is the intent of this section of the metaprogram to provide an operational algorithm to detect such illegal programs before an attempt is made to interpret them. The tests are commonly stated in terms of existence conditions, using the #THERE-EXISTS

operator of SEMANOL applied to the parse tree. The iterator and sequencing operators of SEMANOL, combined with the use of an abstract syntactic representation, allow these restrictions to be expressed succinctly. Since the application of these tests is made before interpretation proper, they can cause the rejection of programs that could be interpreted without encountering the error condition. That is, these tests correspond to those that a compiler might make and the consequences can be semantically different than if similar tests were applied at execution time (e.g., consider unexecuted references to undefined variables). Typical tests included here are those for name uniqueness, type conformity, formal and actual argument agreement, program accord with structural rules, data organization consistency, etc. While SEMANOL enforces these limitations readily, the formulation of rules for syntactic exclusion is a very difficult problem in language design.

Coincident with the imposition of the context sensitive syntactic restrictions, the Ada metaprogram also defines the method by which generic instantiation is to be done. The method used with Ada is new to SEMANOL and is essentially one of traversing the parse tree in a depth first fashion, recursively applying a SEMANOL function to each of the descendant nodes. When a generic is encountered, the body of the generic declaration is specially scanned to produce the string of terminals of the declaration with actual parameters substituted for formal ones and global name references replaced by fully qualified name references. The resulting new program text is then parsed again, using the same concrete grammar as before, so as to provide the parse tree used as the basis for explaining execution semantics.

Having specified these preliminary operations, the semantics of program execution are next given. Figure 3 illustrates the data structures that are used to explain execution semantics. The parse tree is (effectively) a static structure that is traversed and tested by direction of the SEMANOL metaprogram; the SEMANOL metaprogram then causes the execution effect of the parsed program to be realized. The semantics of the SEMANOL specification of a defined language are thus generally expressed in terms of the grammar that corresponds to the concrete grammar used to create this parse tree. The storage mechanism

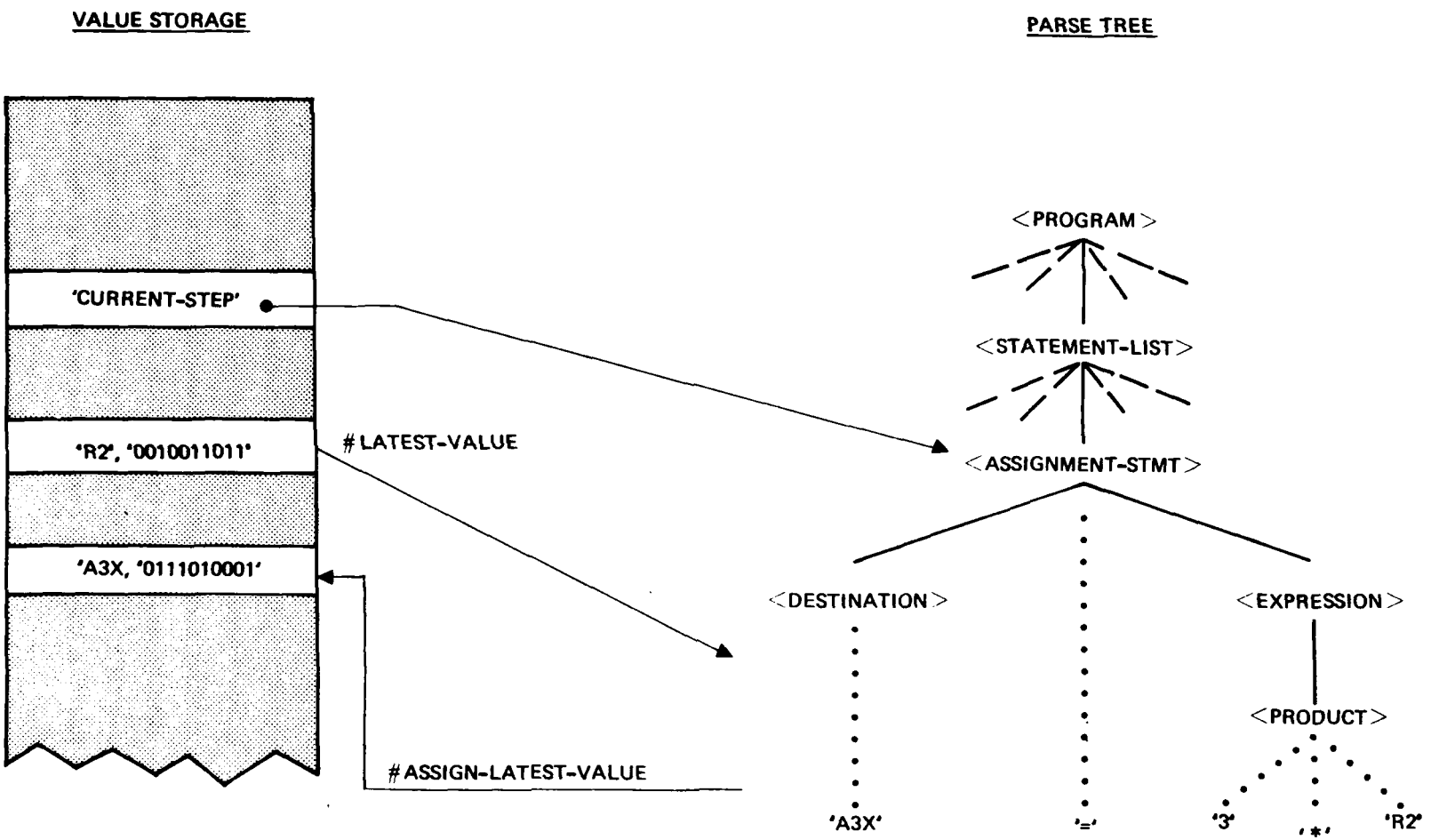
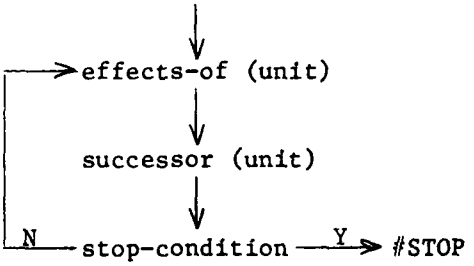


Figure 3: The Data Structures of Interpretation

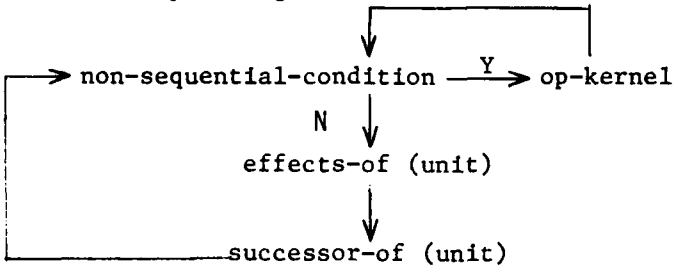
of SEMANOL is used to record the changes that occur as the interpretation is performed. Note that this overall model corresponds rather closely to that of any conventional computer; this similarity emphasizes SEMANOL's operational style of definition.

While there are several interwoven facets to the description of run-time (i.e., dynamic) semantics, it seems best to begin by describing the overall structure that we use in SEMANOL to control run-time interpretation. Essentially, run-time semantics are given with reference to the execution units that are defined for a given language. Execution units will be statements when possible, as it was for Basic but, more commonly, are lesser phrases into which statements must be divided for semantic descriptive fidelity. In the case of Ada, execution units are defined to be very elementary units (such as a sum or product operation) in order that exceptions and interrupts can be faithfully modeled. Based on the execution units chosen, the semantic definition of an execution-unit is decomposed into an effects-of part and a successor part; the effects-of part giving changes to the machine state that execution of the unit causes, with the successor part serving to identify which unit is to be executed next. Note that the elaboration of Ada declarations is treated in this process, not in some separate pre-execution phase, since the semantics of declaration elaboration carry such strong execution implications. Graphically, the control structure has this form for a sequential language;



While this form of control loop is the one developed for past specifications and is fine for describing the sequential semantics of a single task, it is inadequate for Ada because it fails to provide for control transitions between Ada tasks.

For Ada, this control loop is augmented as follows:



Thus the control loop will cycle through the interpretation of the execution units until a non-sequential-condition is activated for this SEMANOL processor. This condition can be set by the task being interpreted, because of an accept or delay for instance, or by another Ada task interacting with this one, because of an abort perhaps. All forms of interrupts, exceptions, and cross-task interaction can be accommodated in this model, and with a resolution similar to that which will exist in practice.

The recognition of a non-sequential-condition causes the op-kernel to be invoked. The op-kernel is largely meant to give the semantics of Ada task scheduling and to complete the effect of control transitions that may involve task scheduling. The scheduling of Ada tasks is essentially a matter of queue management and the shuffling of Ada task identifiers between queues as their status changes (e.g., from a waiting queue to a ready queue when an enabling condition is recognized.). In effect, an Ada task management subsystem is given in a set of SEMANOL functions that closely resemble an operating system kernel. The SEMANOL processes correspond to the hardware processors of a multiple-processor, shared storage, host system. An implementation is thereby created, as needed for a truly operational metaprogram, that is meant to make obvious the issues involved in the semantic definition of Ada concurrency. The number of SEMANOL processes is determined by a static parameter of the metaprogram, and this parameterization emphasizes the implementation dependent character of that aspect of semantic specification.

The successor semantics of sequential languges have been, by convention, stated in terms of the declared SEMANOL global variable CURRENT-STEP. CURRENT-STEP

holds the node of the executable unit (e.g., statement phrase) now being processed. The active point in the defined language program is defined by this value, and transitions of control by changes in this value. Because of parallelism, multiple active execution points may exist in Ada, one in each task. Thus the metaprogram for Ada provides for having a control frame for each task, with one element in each frame being the CURRENT-STEP value for that task. Because SEMANOL process names are appended to the names they use when referencing storage, each SEMANOL process naturally has a set of names effectively local to it. Thus each concurrent SEMANOL process can conveniently maintain its own state variables, such as CURRENT-STEP. The semantics of sequencing within a task, therefore, can be given with much the same techniques as used previously for purely sequential languages. Since these techniques have been proven, their adoption for Ada improves the confidence one may have in the Ada design.

To support the method of modeling Ada concurrency just outlined, it is intended that SEMANOL be extended through the addition of the #CO-COMPUTE, #P, and #V primitives. The #CO-COMPUTE operator activates multiple copies of the SEMANOL semantic procedure containing the control loop illustrated, and so introduces concurrency into the SEMANOL metalanguage. The #P and #V operators add binary semaphores to SEMANOL so that SEMANOL processes can be coordinated and access to shared variables, such as scheduling queues, regulated. The extensions to SEMANOL were thus developed to answer the needs of providing a particular, but generalizable, model of concurrent semantics.

The effects-of semantics are primarily given in terms of the storage model, evaluation rules, and the input-output model. The nature of the storage model used depends upon the language being defined and the level of detail that is being provided. When storage semantics can be divorced from implementation dependencies, storage names directly derivable from the variable names used in the program itself may be used, and values stored at these storage locations can be represented by a convenient syntax. A rather abstract storage model is thereby provided. It is this type of storage model that is contained in our Ada design, and it is one natural to SEMANOL's storage operators. If it were

later decided to include detailed representation specification semantics in the Ada metaprogram, the SEMANOL design could be readily extended to deal with the hard realities of bits and computer words. The effect of extension would be that the names used for storage and retrieval would become (equivalent to) storage addresses and that the values associated with these names would become the binary contents of the addressed locations. A form of storage allocation would thus be defined so that these storage addresses could be established. The SEMANOL metaprogram thereby becomes appropriate only for a given language implementation; however, this condition is inescapable if these semantics are to be completely defined.

Closely related to the storage model is the semantic definition section that explains the evaluation rules for the language being defined. Here are explained the semantics of arithmetic operations of the types (e.g., integer, floating, fixed, location) provided in the defined language, character string operations, Boolean evaluation, and comparison relations. The description of evaluation may include consideration of machine effects upon the computational result produced, type conversions that may be required, and a suitable interface with the storage model. The general evaluation procedure followed is largely one of translating the constants and operators of the defined language to corresponding SEMANOL representations, and then performing the computation upon the SEMANOL constants. Evaluation is conveniently described recursively and SEMANOL's recursive abilities are used advantageously here. The use of binary representations for evaluation is commonly done because (1) implementation dependencies are most clearly revealed with that representation and (2) conventional descriptive documents tend to assume an internal binary orientation.

The semantics of input-output deal with data transmission and possibly with translations between internal and external representations of data. The transmissions required are accomplished by the #INPUT operator and the #OUTPUT operator. In this way, input-output semantics relating to the external world can be modeled. Input-output that is internal to a program in the defined language can be described through the use of sequences and a suitable set of

representation conventions.

Naturally, this general specification structure is tailored to fit the needs of the programming language being defined. It is also subject to the stylistic inclinations of the person(s) writing the SEMANOL code; indeed, a good deal of individuality will be found among the SEMANOL specifications that now exist. The writing of SEMANOL specifications does follow a pattern, but it is far from being a rote exercise. The Ada specification reflects tradition and its developer's predilections.

This discussion of SEMANOL metaprogram structure is only intended to impart a sense of the way in which SEMANOL is used to formulate complete programming language specifications. The programming orientation of this type of description should, nevertheless, be obvious. It should also be clear that this form of specification can become extremely detailed, and when so used, it becomes a language processor that is obligated to define language attributes ordinarily considered to be implementation defined. Languages are defined with allowance for implementation variability because:

1. At the very least, differences are expected to exist among language implementations because of differing data representations, computational incompatibilities, conflicting error treatments, operating system induced discrepancies, etc. The extent of differences among implementations will largely be determined by the range of host environments employed and by how strongly code transferability has influenced the design of a particular programming language, but differences are expected even for languages popularly considered "machine independent".
2. Different language processors are expected to order operations differently and otherwise generate code that is meant to capitalize upon the unique features of a given machine. The code produced may also be influenced by the compiling techniques that are employed to improve compiler efficiency. Computing efficiency is thereby realized but in ways that are hard to describe precisely.

In short, efficiency is more important than strict uniformity. Thus one goal in writing language definitions is to construct a specification that describes the idealized (machine independent part of the) language, while it also clearly identifies features whose meanings are left to be the consequences of a given execution environment. If possible, the implementation dependent parts should be described to the extent that the range of acceptable implementations can be clearly made part of the language defining specification.

While a deliberately incomplete SEMANOL metaprogram intended only for publication can ignore some of these problems, an operational SEMANOL metaprogram must be made complete through some form of precise specification for all the machine and implementation factors that could otherwise be dismissed in a conventional specification method. That is, the complete SEMANOL metaprogram is itself an implementation, and consequently it must include a specification of semantically significant machine features. This is possible with SEMANOL because its data representations and operators are indeed machine independent, and so capable of modeling conventional machine features.

A conscientious effort is than made, in writing SEMANOL metaprograms, to separate these machine details from the other code so that they do not obscure the broader semantic specification. Historically, machine features such as word size, arithmetic operations, and overflow treatment have been described by the SEMANOL metalanguage in a distinguished part of the metaprogram that is meant to describe these machine-dependent elements of the semantics. Standards for similar computers can then be prepared by revising the SEMANOL code that expresses these machine dependencies. In fact, a strong effort has been made in the past to parameterize these features so that a family of metaprograms can be built, each differing from the other only in the values given to these machine dependent parameters.

It should also be noted that machine details can oftentimes be given in external functions. The SEMANOL metalanguage supports this option through the #EXTERNAL-CALL-OF feature. As a result, a library can be built that contains

routines to simulate the hardware and operating system functions that are needed to complete the description of the language being defined. Implementation dialects can then be distinguished by separate libraries. Parameterization and external functions are both used in the design of a SEMANOL specification of Ada.

The SEMANOL Metalanguage

Now the semantic operator for L, P_L, could certainly be written in conventional programming languages, such as JOVIAL or Fortran; however, other programming languages were not designed with formal semantic description in mind. Therefore, semantic interpreters would be difficult to write in these languages and, more importantly, the interpreters so expressed would be very difficult to understand. This lack of comprehensibility means these interpreters would serve poorly as specification standards. Contrarily, SEMANOL is a metalanguage specifically designed, and repeatedly refined, for expressing the semantics of programming languages. Because of this, an interpretive specification stated in SEMANOL is relatively easy to understand; the keywords and structure of SEMANOL, coupled with the use of proven specification conventions, provide a "naturalness" to the SEMANOL metaprogram that is not available with other interpreters. Precision is therewith combined with relative readability through the use of SEMANOL.

One part of the SEMANOL metalanguage must deal with syntactic forms since SEMANOL expresses the rules by which programs and input are given in concrete format. Thus SEMANOL contains syntactic definitions that are combined to form context-free grammars. The SEMANOL syntactic definitions used to define a grammar look much like the productions of the usual treatments of such programming language grammars. For instance,

```
#DF Program =  <'START'> <#GAP> <Statement-list> <#GAP>
               <'TERM$'>#.

#DF Statement-list =  <Statement> <%<<#GAP><Statement>>>#.
```

describes the syntax class Program as being initiated with a START, terminated

with a TERM$, and containing at least one Statement. The #GAP set constant of SEMANOL is context sensitive; it is meant to describe the conventional rules that govern the use of blank characters in programming languages. The % states that zero or more occurrences of <<#GAP> <Statement>> may follow. The grammar so defined is used by the #CONTEXT-FREE-PARSE-TREE operator to construct a derivation parse tree for a given string relative to this grammar. The nodes in a SEMANOL parse-tree are ordered by traversing it in preorder. This corresponds to a left-to-right ordering of substring occurrences. Each non-terminal node is labeled with its syntactic class name, its segment number, and its case number. These labels indicate how the node was obtained by the parser. In lexical terms, the labels describe the syntax of the string of terminal letters in the subtree of which the node is the root node. Note that the #CONTEXT-FREE-PARSE-TREE operator is a generalized parsing algorithm that imposes no artificial restrictions upon the grammar with which it is used.

Besides the node type created by application of the #CONTEXT-FREE-PARSE-TREE operator, SEMANOL contains a variety of conventional and not-so-conventional types that, collectively, fit its specialized field of application. These types are shown in Table 1 in conjunction with the operators that SEMANOL supplies for each type. While the types and operators are generally familiar, and so should be roughly understandable without explanation, a few comments should make SEMANOL somewhat clearer. These comments are organized by type, and are as follows:

1. The Boolean type consists of #TRUE and #FALSE, and the Boolean operators have the conventional interpretation.

2. The bit-string type consists of indefinite length strings of zero's and one's; these operators likewise are to be conventionally interpreted.

3. The integer type is unusual in that integers are numeric strings upon which string arithmetic is performed. Thus the range of SEMANOL integer arithmetic is unaffected by underlying machine factors and an idealized arithmetic is thereby realized. Note that division

produces a truncated integer result, #NEG is a negation operator, and #CONVERT translates integers from one base (2, 8, or 10) to another. No other arithmetic is provided since, when writing formal specifications, our experience has been that floating point (and fixed point) semantics are best modeled through the use of integers. The high-level iterators permit search algorithms to be stated in an obvious and understandable way.

4. The string type is largely conventional, but it does provide facilities so that ordinarily non-represented characters, such as end-of-line tokens, can be conventionally included. The #CW operator performs string concatenation, with the meaning of the other operators being fairly self-evident.

5. The sequence type is composed of groups of ordered elements of any type; a sequence need not be homogeneous. Since the elements of a sequence may themselves be sequences, any form of hierarchical structure is easily modeled. The names of the sequence operators are meant to be self-descriptive except for the concatenation operator, #CS.

6. The node type is produced initially by the #CONTEXT-FREE-PARSE-TREE operator as explained previously. The #PARENT-NODE and #ROOT-NODE operators traverse a parse tree in an upward (i.e., rootward) direction, while #SEG moves in a downward direction. #SEG-COUNT returns the number of immediately descendent nodes for a given node. The #SEQUENCE collectors scan the parse tree in a uniform way so as to generate a preordered sequence.

Table 1: SEMANOL Operators

[Notation: b=bit-string, e=any expression, i-integer, ℓ=logical, n=node, s=sequence, w=string, sc=syntax class, x=dummy]

Boolean Type

ℓ1 #AND ℓ2

ℓ1 #IFF ℓ2

ℓ1 #IMPLIES ℓ2

ℓ1 #OR ℓ2

#NOT ℓ

Bit-String Type

b1 #BAND b2

b1 #BOR b2

b1 #BXOR b2

Integer Type

#ABS(i)

#CONVERT 2/8/10 (i)

#NEG i

#SIGN i

+, -, /, *

#FIRST x:i1<=x<=i2 #SUCH-THAT(ℓ)

#LAST x:i1<=x<i2 #SUCH-THAT(ℓ)

String Type

w1 #CW w2

#FIRST-CHARACTER-IN w

#LAST-CHARACTER-IN w

#LEFT i #CHARACTERS-OF w

#RIGHT i #CHARACTERS-OF w

#LENGTH(w)

#PREFIX-OF-FIRST w1 #IN w2

#SUFFIX-OF-FIRST w1 #IN w2

#SUBSTRING-OF-CHARACTERS i1 #TO i2 #OF w

#SUBSTRING-POSIT-OF w1 #IN w2

i #TH-CHARACTER-IN w

Sequence Type

s1 #CS s2

#FIRST x #IN s #SUCH-THAT (ℓ)

#LAST x #IN s #SUCH-THAT (ℓ)

i #TH x #IN s #SUCH-THAT (ℓ)

#FIRST-ELEMENT-IN s

#LAST-ELEMENT-IN s

i #TH-ELEMENT-IN s

#INITIAL-SUBSEQ-OF-LENGTH i #OF s

#TERMINAL-SUBSEQ-OF-LENGTH i #OF s

#LENGTH (s)

#ORDPOSIT e #IN s

#REVERSE-SEQUENCE (s)

#SUBSEQUENCE i1 #TO i2 #OF s

#SUBSEQUENCE-OF-ELEMENTS x #IN s #SUCH-THAT (ℓ)

Node Type

#PARENT-NODE (n)

#ROOT-NODE (n)

#SEG i #OF n

#SEG-COUNT (n)

#SEQUENCE-OF <sc1> #U<Sc2>...#U<scn> #IN n

#SEQUENCE-OF-ANCESTORS-OF n

#SEQUENCE-OF-NODES x #IN n #SUCH-THAT (ℓ)

#SEQUENCE-OF-NODES-IN n

#STRING-OF-TERMINALS-OF(n)

A few SEMANOL operators are not easily classified and do not appear in Table 1. #ASSIGN-LATEST-VALUE and #LATEST-VALUE are operators that store and recall information from what is, in effect, a single level associative memory wherein information is stored as (name,value) pairs. The names used here are ASCII strings of any length; the values may be any SEMANOL expression value, and so may be complex sequence or node structures as well as simple integers or bit-strings, for example. This abstract storage structure of SEMANOL permits ready modeling of any storage model needed when defining storage semantics of a programming language. #GIVEN-PROGRAM reads a string, normally the defined language program to be interpreted, from one file, while #INPUT reads strings, commonly data input to the program being interpreted, from another file. #OUTPUT is a general purpose string output operator. #STOP and #ERROR are used to variously terminate the interpretation process.

The relational constants of SEMANOL are shown in Table 2 and are generally the expected ones, with natural extensions of equality and inclusion to parse tree nodes and sequences. An ordering relation, #PRECEDES, also applies to sequences and parse tree nodes. There are two quantifier relations included within SEMANOL,

#FOR-ALL...#IT-IS-TRUE-THAT...
#THERE-EXISTS...#SUCH-THAT...,

that are very useful. The general inclusion of high level iterative facilities throughout SEMANOL permits expressive clarity to be provided the reader, and also aids in the preparation of specifications.

The rules for expression composition in SEMANOL are typical of those found in contemporary programming languages.

SEMANOL metaprograms are then composed of semantic definitions and semantic procedures. A semantic definition is similar to a function declaration and has the following possible formats:

#DF $f(v_1,\ldots,v_n)$ => exp #.

or

Table 2: SEMANOL Relations

Bit-String: #EQ, #NEQ

Integer: <,<=, =, #N=, =>,>,

#FOR-ALL x: i1<= x<=i2 #IT-IS-TRUE-THAT(ℓ),

#THERE-EXISTS x:i1 =x<=i2 #SUCH-THAT(ℓ)

String: #EQW,#NEQW,#IS

Sequence: #EQW,#NEQW,#IS-IN,#PRECEDES,

#DOES-NOT-PRECEDE,

#FOR-ALL x #IN s #IT-IS-TRUE-THAT (ℓ)

#THERE-EXISTS x #IN s #SUCH-THAT (ℓ)

Node: #EQN,#NEQN,#PRECEDES,#DOES-NOT-PRECEDE,

#IS#NODE-IN,#IS-#NOT#NODE-IN,

#IS <sc> , #IS#CASE

```
#DF f(v1,...,vn)
    => expl #IF B1;
    :
    => expq #IF Bq;
    => expo #OTHERWISE #.
```

Here, f is the name of the function being defined, $v_1,\ldots,v_n$ are dummy parameters serving as the variable arguments of f (there may be no parameters), and the expi are SEMANOL expressions built up from constants, global variables, the parameters v_i and possibly references to other functions defined elsewhere in the metaprogram. The B_i are boolean expressions, only one of which should be true for any particular set of actual values of the v, that determine which exp will be evaluated to become the value of f. Semantic definitions can be applied recursively without restriction (apart from common sense).

In addition to the semantic definitions, SEMANOL includes facilities for writing sequential programs composed of commands (or statements). Commands are grouped together in semantic procedures and used to form the #CONTROL-COMMANDS section of a metaprogram. A semantic procedure is denoted by a #PROC-DF of the general form

```
#PROC-DF f(v1,...,vn)
```

where f is the procedure name and the v_i's are parameters. Procedures can return values, change the values of arguments, and affect the computational state as a side-effect of execution. The sequential listing of commands allows the operations they denote to be synthesized into sequential algorithms in the usual manner of programming languages. However, the SEMANOL commands are few and simple. The structures for synthesizing metaprograms are also fairly simple and ones known to most programmers. Commands are built up recursively starting with three atomic commands:

```
#COMPUTE! exp              (Evaluate expression exp.)
#ASSIGN-VALUE! x = exp     (Assign value of exp to x.)
#RETURN-WITH-VALUE! exp    (Terminate #PROC-DF, returning value of exp.)
```

Here, exp is any SEMANOL expression and x is a declared global variable. The synthesis of commands into composite commands is done recursively with the following synthesizers:

#BEGIN $c_1c_2...c_n$ #END	(Causes the sequence of commands $c_1,...,c_n$ to be treated as a unit.)
#IF Bexp #THEN c	(Bexp is a boolean expression. If it is false, command c is skipped.)
#FOR-ALL i: m<=i<=n #DO c	(Iterative loop. Execute command c for $m \leq i \leq n$, incrementing i by 1 each execution.)
#FOR-ALL i: m<=i #DO c	(Iterate c for $m \leq i$.)
#FOR-ALL i #IN seq #DO c	(seq is a sequence, say $(x_1,...,x_n)$. Execute c for $i=x_1$, $i=x_2,...,i=x_n$.)
#WHILE Bexp #DO c.	(Execute c as long as Bexp is true.)

For standardization of sequential algorithmic programming languages, these have proved to be effective. They define the semantics of "control" (i.e., sequencing of statement execution) in such languages in detailed precise operational terms, using simple programming constructs that are well understood.

SEMANOL Extensions for Ada

Prior to this contract, SEMANOL had been used to formally describe programming languages that were purely sequential in their control semantics and so devoid of parallel execution. SEMANOL, as just described, worked admirably in these cases. However, to model the concurrency of Ada, as decided upon from our analysis of Ada semantics, required that SEMANOL itself be extended to include a form of parallel SEMANOL process execution. By adopting this approach, the execution of concurrent Ada tasks can be simulated much as they might occur on a multiple processor host computer, and a clear picture of the implications of parallelism thereby presented. It is only for the addition of concurrency that SEMANOL changes are needed.

Before describing the additions required by our approach, it may be helpful to describe a few changes that were not made to the metalanguage but provided as external functions. SEMANOL's #EXTERNAL-CALL-OF feature permits operational extension of the SEMANOL metaprogram to procedures defined outside the metaprogram. Such external functions must exhibit certain characteristics, but the detailed algorithm is not considered part of the formal specification. The #EXTERNAL-CALL-OF feature is infrequently used, but it has been employed as a device to graphically isolate programming language semantics considered to be implementation defined. For Ada, a *time* and *choice* function are presumed. The *time* function is used to provide some form of value that can provide support when defining the CLOCK, DELAY, and SECONDS semantics of Ada. The *choice* function is used by the SEMANOL-expressed scheduling algorithm for Ada tasks to resolve scheduling choices that are arbitrary to the Ada semantics. Either could certainly have been added to SEMANOL as a new primitive; the choice not to do so reflects the implementation dependent nature of each, and so the difficulty of generalizing their definitions, as well as a desire to minimize the extent of SEMANOL changes. Indeed, the way in which either is eventually provided makes little difference since a SEMANOL interpretive implementation cannot meaningfully correspond to a conventional implementation where timing is a factor. The implementation dependent nature of these functions is emphasized by their relegation to external functions.

The changes to SEMANOL(76) that the Ada specification design presumes are these (presented much in the style of the current metalanguage reference manual):

1. The inclusion of a facility by which concurrent SEMANOL tasks may be initiated. This is the new #CO-COMPUTE! command. Its syntax is given by

```
#DF simple-statement => <'#CO-COMPUTE!'> <gap>
    <Semantic-def-name> <gap> <'#WITH-NAMES'> <gap>
    <'('> <gap> <sequence-expression> <gap> <')'> #.
```

Given

#CO-COMPUTE! a #WITH-NAMES (b_1,...,b_n) #.

returns

<error> if a is not a semantic definition taking zero arguments.
<error> if b_1,...,b_n are not string convertible.
<error> if there is currently an active task with name b_1 or b_2 or .... b_n.

otherwise: A set of n SEMANOL tasks is created, with each task body consisting of the semantic definition a as defined by a #DF or #PROC-DF. These tasks are given the names b_1,...,b_n. These SEMANOL tasks may be executed concurrently with each other and with the task issuing the #CO-COMPUTE!, but the number of such tasks actually running concurrently, their relative interpretation rates, the nature of task scheduling, etcetera are not defined, except that processes cannot be delayed indefinitely. Nevertheless, an Interpreter option will be provided that will impose an arbitrary deterministic order on the interpretative cycle (thus creating a repeatable trace) so that SEMANOL specification testing and debugging can be more easily done. The SEMANOL tasks are initiated as though by a conventional call invocation without parameters. Tasks are terminated, and so removed from the SEMANOL system and their names re-released, when they complete normally or invoke #STOP or #ERROR. Normal termination occurs when the selected expression is evaluated for a #DF or a #RETURN-WITH-VALUE is interpreted in a #PROC-DF. The values returned in both cases are disregarded (but should be #NIL for the sake of clarity). The execution of #ERROR causes all tasks to terminate; but the state of the SEMANOL machine on an #ERROR termination cannot be predicted in the face of concurrency. #STOP likewise causes all tasks to be stopped, and will return an error condition if any other SEMANOL task exists within the system.

In view of this SEMANOL extension for concurrency, the nature of the #CONTROL-COMMANDS section has changed slightly. The metaprogram will still start execution with the first command of the #CONTROL-COMMANDS, but the #CONTROL-COMMANDS section is now considered to be executed by a SEMANOL task with name #NIL. Note that this extension has no effect upon existing SEMANOL specifications.

2. The addition of features by which parallel SEMANOL tasks can be coordinated. We have found integer semaphores to be adequate for our approach to modeling Ada concurrency, and have adopted these because of their relative simplicity and wide familiarity within the programmer community. Syntactically, semaphores are included in SEMANOL by adding the syntax class Concurrency-statement to Simple-statement and defining Concurrency-statement as follows:

```
#DF Concurrency-statement
    =><'#V!'><gap><String-expression>
    =><'#P!'><gap><String-expression>#.
```

The semantics of semaphores are conventional and given as follows:

#V! a

<error> if a is not string-compatible.

otherwise: The value of the integer variable a is incremented by one, with the reference, addition, and storage of a being a single indivisible operation. This is equivalent to the indivisible execution of the following:

```
#COMPUTE! V(A)      where

#DF V(a)
  => #ASSIGN-LATEST-VALUE(a,#LATEST-VALUE(a)+1)#.
```

#P! a

<error> if a is not string-convertible.

otherwise: The SEMANOL process giving the #P is delayed on this operation until the integer variable a becomes greater than zero. The nature of the wait is unspecified and carries no scheduling or queueing implications. When the test is passed, the value of a is reduced by one coincident with the process continuing. In SEMANOL terms, this is

```
#COMPUTE! P(a)

#PROC-DF P(a)
#BEGIN
  #WHILE test-and-set(a) = 0 #DO null-action
  #RETURN-WITH-VALUE  #NIL
#END #.
```

where test-and-set is the following indivisible action:

```
#PROC-DF test-and-set(a)
#BEGIN
  #ASSIGN-VALUE! temp = #LATEST-VALUE(a)
  #IF temp > 0 #THEN
     #COMPUTE! #ASSIGN-LATEST-VALUE(a, temp-1)
  #RETURN-WITH-VALUE! temp
#END #.
```

The null-action can be any statement that does not semantically affect the state of the SEMANOL machine. Note that #V! and #P! are commands, and so appear in #PROC-DF's or the #COMMAND-CONTROL section.

3. The augmentation of the references to declared variables so that the name of the SEMANOL process making the reference will now be made a prefix to the declared name. Thus a reference to a SEMANOL global variable is now taken as an abbreviation for the semantic expression #LATEST-VALUE(name #CW 'x'), where name is the name of the SEMANOL process evaluating the expression. The statement

   #ASSIGN-VALUE! x = semantic-expression

   is an abbreviation for the statement

   #COMPUTE! #ASSIGN-LATEST-VALUE(name #CW x, semantic-expression).

   This change is a direct consequence of the style used with SEMANOL in that declared variables are ordinarily used when describing control semantics. With concurrency, a set of such variables will be needed by each SEMANOL process and each set is conveniently qualified with the SEMANOL process name to distinguish it from the others. By adopting this convention, we also believe descriptive clarity is gained. Because of this convention, there is no operator available by which a SEMANOL process may obtain its name.

The SEMANOL changes can thus be seen to be few in number and without affect upon existing SEMANOL metaprograms (e.g., JOVIAL or BASIC).

The Operational SEMANOL System

The specification of a programming language expressed in the SEMANOL meta-language is made into an operational system by the SEMANOL Interpreter. The SEMANOL Interpreter accepts a SEMANOL specification of a programming language and uses that input specification to realize the semantic effect of (i.e., to execute) programs written in the language thus defined. By virtue of the Interpreter, SEMANOL specifications can themselves be tested and debugged. Furthermore, an operational standard for the defined language is thus created.

The operation of the elements that constitute the SEMANOL system is shown in Figure 4. The broken line encloses the SEMANOL Interpreter, which can be seen to actually consist of two loosely connected programs identified as the Translater and the Executer. The Translator processes the SEMANOL description of a programming language and so generates an intermediate code form known as the SEMANOL Internal Language (SIL) file. This translation phase also tests the SEMANOL description for its syntactic correctness, much as a conventional compiler would do, and continues only for acceptable descriptions. The SIL representation corresponds to a list of operators, operands, and direct transfers that are used to control the Executer. The Executer program is essentially a stack oriented processor that then interprets the SIL code and so performs the operational interpretation of a program in the defined language. As shown in Figure 4, this interpretive processing commonly includes the reading of input and the production of output at the direction of the defined language program, p, being processed.

A set of user commands provide the means by which the Interpreter programming system can be directed and by which the features of incremental translation and testing can be controlled. The nature of incremental translation and testing control are explained in what follows.

Incremental translation provides a means by which a metaprogram presently being processed by the Executer can be dynamically modified. Incremental translation is invoked by commands during execution; it causes user-supplied SEMANOL metalanguage statements to be processed by the Translator, as if part of the active metaprogram, and then merged into the metaprogram being processed. Since only changed statements are translated, rather than the entire metaprogram, computer processing time can be reduced during a testing session. As the computational state at the time of incremental translation is saved by the Executer, it is often possible to continue the computation with the new metaprogram without repeating the processing to the incremental translation point. Thus further time savings are possible. Since SEMANOL metaprograms are often large, since metaprogram interpretation is intrinsically

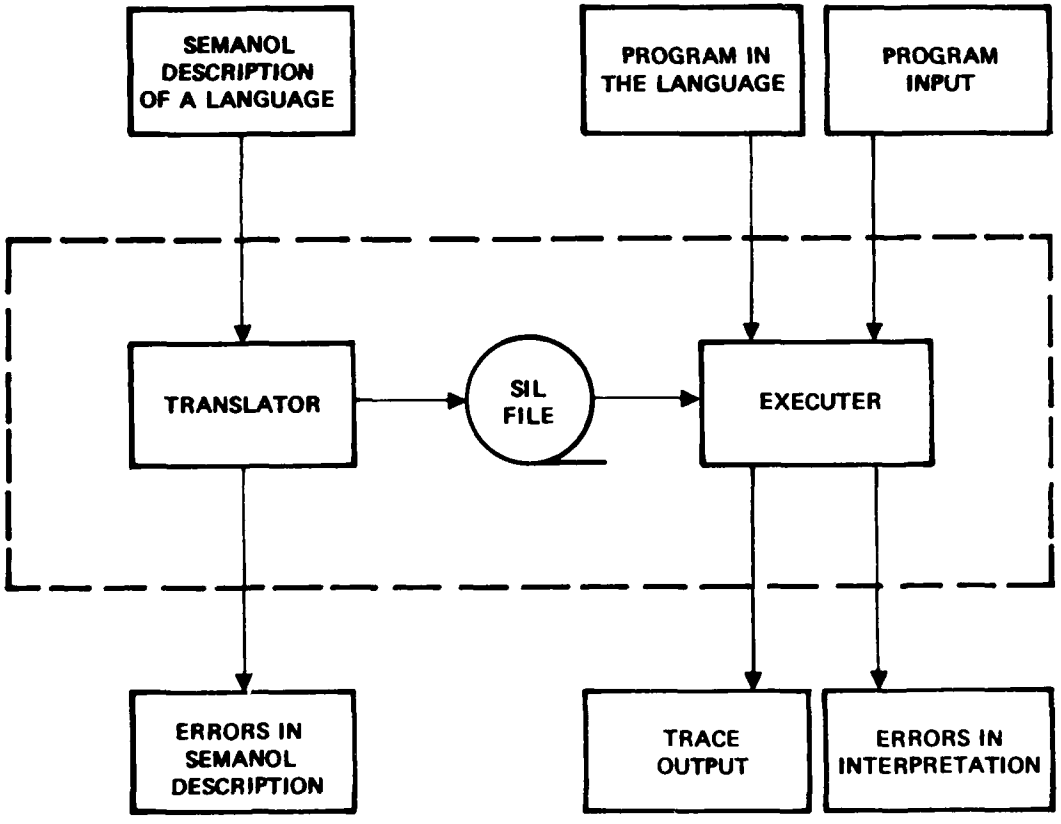


Figure 4: The SEMANOL Interpreter

rather slow, and since processing is ordinarily done interactively from remote sites, the savings provided by incremental translation are very helpful.

The testing features give the user control over the information provided during SEMANOL metaprogram execution and the ability to interact dynamically with the running metaprogram. The test features fall into two classes:

1. Trace features, that are provided so that control flow through a SEMANOL metaprogram can be followed. The trace provides a time-ordered sequence of semantic definition (i.e., function) invocations and returns. This trace can be directed to a file of the user's choice. The user is also given wide control over the content of the trace through several commands that permit naming the specific definitions to be traced and controlling whether subsidiary definitions are traced or not. Trace volume, which can become overwhelming, can thereby be restricted and the test process itself facilitated since the user need investigate only trace information significant to the immediate problem. The current trace status is always available to the user.

2. Break features, that are provided so that user interaction with the running metaprogram can be obtained. The essential characteristic of this feature is that the user can establish points within the metaprogram at which processing will be suspended and control relinquished to the user. The user can then interrogate the state of the computation or otherwise interact with the running metaprogram. Breakpoints are set on semantic definitions, and can be easily established or removed by the user through Executer action. Breakpoint status is also readily available to the user. The user action upon a break, apart from altering the break conditions themselves, is accomplished by the processing of SEMANOL statements. Such statements can be interactively introduced by use of the incremental translation feature or initially included in the

metaprogram to provide this run-time support. This code is then executable upon user command at the break. Thus the SEMANOL metalanguage also serves as the language of user interaction. Associated with this break procedure is an interrupt option that uses the escape mechanism of Multics to provide a user with the ability to suspend Executer processing before the next semantic definition is interpreted. That is, a break is forced at the next convenient point in the metaprogram. A high-degree of user control is supplied through the break features.

In all, a series of twenty-one user commands is provided. The command set provides the user with a convenient way in which to direct the SEMANOL Interpreter programming system, and makes available an interactive test facility that corresponds closely to those available with the better conventional programming language support systems. The effectiveness of the SEMANOL Interpreter for users is therby increased.

The Interpreter was not modified in performance of this contract as that was not within the scope of work. Thus the SEMANOL metalanguage extensions defined earlier are not yet operational. Their implementation would obviously be required were an operating SEMANOL specification metaprogram for Ada to be created in the future. Besides the metalanguage changes themselves, certain other changes to the Interpreter would also be expected to accompany the implementation of the new SEMANOL concurrency primitives, to wit:

1. The trace output would be augmented so that each trace message would include the name of the SEMANOL process responsible for its generation.

2. The name of the SEMANOL process would likewise be added to each break message initiated by that process.

3. While trace and break output would be associated with a SEMANOL task, it is expected that the setting of trace and break conditions would not be changed. Thus they would be effective for all SEMANOL tasks, rather than for only a stipulated task or group of tasks.

4. While the semantics of the SEMANOL metalanguage do not call for parallel SEMANOL processes to duplicate their execution order from one run to the next, an option would be provided that would cause a predictable execution path to be followed. The repetition of a pre-determined order would aid metaprogramming debugging by giving consistent results for parallel processes, insuring trace repeatability, and providing a sequence of break states that are likewise historically comparable.

With these modifications, the existent Interpreter would be naturally extended to support concurrent metaprogramming.

AMBIGUITIES IN ADA

In analyzing the Ada Reference Manual [21] during contract performance, a great number of questions about the intended nature of Ada arose that we ultimately could not resolve from the information given in that Manual. In preparing our design, we have had to choose solutions to some of these questions as noted in our reports. Other questions did not affect our design, because their level of detail was lower than that of our design in these areas, and so were otherwise disregarded. The purpose of this section is to list a representative collection of these definitional problems that we found. Unless otherwise noted, page references are to [21].

1. There is an ambiguity that makes it impossible to distinguish character-strings of one character from one-character literals in many possible cases. The syntax of each is the same, and the context cannot always be used to resolve the ambiguity. For instance,

```
procedure x(y:in character) is ...;
procedure x(y:in string) is ...;
 . . .
x("A") ;
```

   leads to confusion about which x is meant in the call statement.

2. The interchangeability of " and % as string brackets is unclear. There is a suggestion in page 2-3 that only one or the other can appear, but that is not an explicit statement. Otherwise, the desired meaning of such forms as "A%B", "A%, and %"% is nowhere stated.

3. The values of the attributes 'FIRST and 'LAST when applied to types with empty ranges appear to be unspecified. (Empty ranges are explicitly allowed, see page 3-5.) Note that the answer here will also affect 'ORD.

4. It is unsaid which label on a loop-statement is to be matched by an identifier in the end loop statement. For example, is

```
<<L2>> <<L1>>
  loop
    :
    :
  end loop L1
```

correct, as we have assumed, or should L2 appear in the end loop statement?

5. The aliasing restrictions for subprogram calls are vague. In particular, it is not said whether such restrictions are meant to be enforceable at compile-time or are such as to require run-time checking.

6. It is unclear whether or not any subrecord of an access record can be of variant type and, if so, whether its discriminant can be altered. The specification of page 5-2 is too brief to cover this issue fully.

7. The order of evaluation for a given precedence level is specified to proceed in textual order from left to right on page 4-6 (line 1). However, it is stated that function calls in expressions (because of the lack of side-effects) can be evaluated in any convenient order on page 6-5 (paragraph 2). A contradiction thus exists, and the rules for evaluation ought to be made precise.

8. On page 6-4 it is stated that a subprogram specification given in its body must be identical to the specification given in the corresponding subprogram declaration, if both appear. Unfortunately, "identical" is not explained: it might mean lexically identical, semantically equivalent, lexically identical and appearing in equivalent environments, etc. One needs a rule that is precise enough to determine when the intended effect of this rule has been satisfied (since lexically identical texts can appear in environments that are different and so lead to results we would think are undesired).

9. The point at which the NO-VALUE-ERROR exception is raised when attempting to leave a function without giving a return statement is not given. If raised in the function, it can be handled therein; otherwise, it is the handler in the calling program unit that will be invoked.

10. It appears that the syntax of generic instantiation is wrong, and ought to use new designator instead of new name. The changed syntax would then allow instantiation of defined (overloading) generic operators.

11. The syntax of page 12-1 for generics appears to allow (1) generics to be nested and (2) for generics to be used as formal arguments to generics. However, this seems to be unintended or, if intended, unexplained. Clarification of these issues is needed.

12. It is unclear whether or not labels may be redeclared; if they can, then the scope of a label is unclear. For instance, given

```
<<A>> declare
      begin
           goto A
           . . .
           <<A>> declare
                 begin
                 . . .
                 end
      end
```

which label is the destination of the goto statement (assuming this is legal)? The rules of page 8-2 seem to support multiple answers to that question.

13. The point at which a package body is elaborated is unspecified. Since elaboration involves initialization, this is a question whose answer has semantic significance.

14. The semantics of unblocking a task T blocked on two or more open alternatives, one being an accept and another a delay, is ambiguous. On page 9-7, it is stated that "an open alternative starting with a delay statement will be selected if no other alternative has been selected before the specified time interval has elapsed." First, notice that the converse must also hold; if an accept alternative has been selected before wake-up time, then the delay alternative must not subsequently be selected. Now, the implementation of entry-call in the

Rationale illustrates the possibility of a timing ambiguity. In this implementation, when an entry-call is made, the called task is removed from the delay-list without checking the clock on the assumption that if it is on the delay-list when the entry-call is made, then wake-up time has not yet occurred. However, because of a queue on a delay-list semaphore, which we claim is required, it is possible that the entry-call is made after wake-up time, in which case the delay alternative should be selected since "no other alternative has been selected before the specified time interval has elapsed". To be consistent with the Rationale's implementation, the semantics of select would have to be stated as follows: "An open alternative starting with a delay will be selected at some time t after wake-up time if and only if no other alternative has been selected prior to t." It seems unlikely that this is the intent embodied in the manual, which seems inclined to give preference to the delay option after wake-up time.

15. The precise timing of exception raising is unspecified in many cases, and this can have significant semantic effect. For example, the Reference Manual discussion of the initiate statement does not specify when the exception INITIATE-ERROR is raised in the initiating task; in particular, if there are inactive tasks in the task list as well as active ones, it is unclear whether all of these must be initiated, or perhaps only those which appear before the first active task, and whether all initiations need take place before the raising of the INITIATE-ERROR.

16. The interaction between raising FAILURE exceptions in other tasks and their performance is unclear. Clearly, there are difficulties in dealing with unrelated rates of execution of separate tasks which will remain intractable. But even within a task, there are unspecified circumstances. For example, since binding a raised exception to its handler may involve many synchronization steps, and thus is assured to be an interruptable process in some cases, it is uncertain whether raising FAILURE must be always delayed until such binding is completed for a previously raised exception, or whether binding to a FAILURE handler may "interrupt" and take precedence over the prior binding process.

17. Whether certain exceptions may even be required to be raised or whether "new", surprising,exceptions may suddenly appear in expression evaluation is uncertain, since expression reordering by the Ada processor (say, compiler) is permitted in at least some cases; see paragraph 2, page 6-5.

18. The wording of the Ada manuals can be interpreted to mean that generic parameters are passed using copy semantics, or reference semantics, or macro-substitution semantics. Macro-substitution is the most likely semantics, and the one we adopted, but if it is used then it is nearly impossible for a compiler to share code between instantiations. Further, users of a generic program unit must be aware of the definition in order to avoid undesirable effects.

19. Several syntactic problems were discovered. The syntax production for character-literal was missing. The grammar for declarative-part was ambiguous, allowing distinct derivations for a module-specification as (1) a declaration or (2) a body.

    The grammar for expressions was ambiguous, allowing distinct derivations for a single parenthesized identifier as (1) a parenthesized expression or (2) an aggregate. The use of the type-mark prefix to indicate both type specification and type conversion was troublesome.

    Several structures were misleadingly allowed in the grammar but excluded in the text; for example "delta simple-expression [range-constraint]" appears where "delta simple-expression range-constraint" is more accurate. Not true ambiguities, these grammatical deceptions often simplify the formation of the abstract grammar at the expense of general comprehension.

20. Since our interpretation of the semantics of the abort statement had a critical effect on the design of the SEMANOL specification of Ada, we list this issue here. The last paragraph of page 9-9 indicates that the abort statement "causes the unconditional asynchronous termination" of the indicated tasks. The key points in our interpretation of this description are: the moment of termination is not fixed, but the aborted task cannot indefinitely delay its own demise. Thus the

underlying model must allow for the preemptive termination of a task which is, say, ignoring its environment, perhaps in an infinite loop. We feel this interpretation arises naturally, but may not be universally taken - thus we question whether the phrase is sufficient to avoid ambiguity.

CONCLUSIONS

This project was successful in achieving its goal of creating a design for a SEMANOL specification for the Ada programming language. Many of the techniques used in prior SEMANOL efforts in formal specification were rather directly transferable to Ada, so this project can be viewed as a natural progression from prior work. At the same time, the need to deal with Ada's form of task concurrency caused us to extend the SEMANOL system so that it would incorporate our approach to modeling the semantics of concurrency. This extension to the semantic model, in turn, required that additions be made to the SEMANOL metalanguage; however, the additions were few in number and so consistent with the existing metalanguage that upward compatibility was able to be maintained. In sum, we were able to use many proven SEMANOL techniques with Ada because the SEMANOL extensions for concurrency were designed to retain a maximum amount of consistency with the existing approach and metalanguage. By using previously successful methods, our confidence in the Ada design has been strengthened.

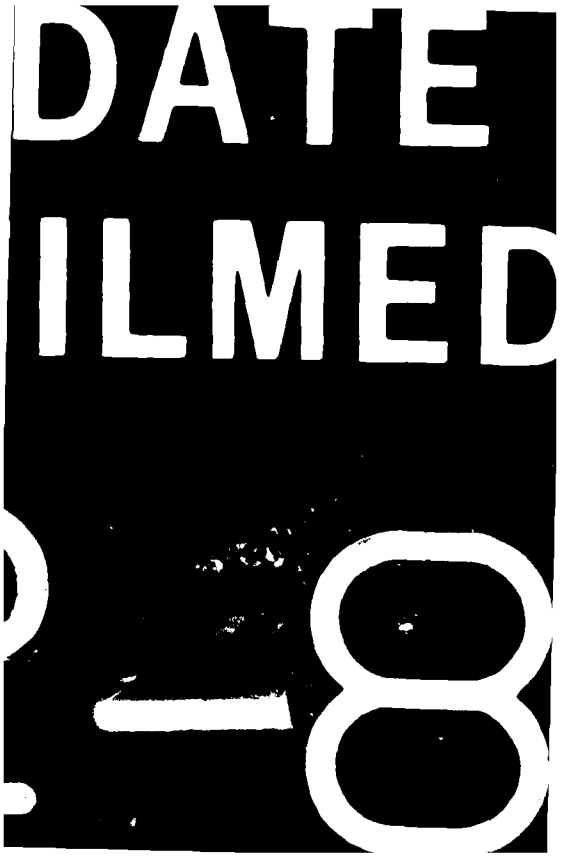
Additionally, the ability to borrow from the results of earlier projects meant that greater attention could be given to the novel features of Ada. The major consequence of this was that added emphasis could be given to developing a SEMANOL approach to modeling the semantics of parallel execution. We have developed a practical approach that seems more satisfying than other proposed methods. A practical design basis has thus been established from which a complete, operational, SEMANOL specification of Ada can be implemented.

Implementation of the SEMANOL specification of Ada from TRW's design would create a formal operational definition for Ada. This type of Ada definition would appear useful in an integrated program to promote the acceptance and use of Ada in the near future. The practical, operational, flavor of a SEMANOL specification of Ada recommends its use in efforts to develop hardware architectures for efficient Ada program execution as well as the corollary efforts to design supportive software environments for Ada program development. The critical issue of an intermediate language would benefit from the application of SEMANOL techniques since (1) a precise definition of the intermediate

language could be created and (2) its relationship to the Ada language formalized. In this and other ways, the availability of a formal SEMANOL specification of Ada can support Ada's orderly development. The time is appropriate to introduce formal semantic descriptive methods into the programming language development and control process; this project's results can contribute to that.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DATE
ILMED